The CE218 Assignment Report for InconvenientSpaceRocks

# Manual

Ok so basically there are some rocks and they are being very inconvenient and also they’re in space pls shoot them for us also pls don’t die k thx bye

* What you need in order to play the game
  + A computer that can run Java
  + A keyboard
  + A mouse (optional)
  + Speakers (optional)
  + A monitor (optional, but highly recommended)
  + The files for the game itself
* Running the game
  + Import the folders from this Java project into a Java project in the Java IDE of your choice
    - These folders are the ones which are actually necessary
      * /images
      * /SaveData
      * /sounds
      * /src
      * /textAssets
  + Build and run it
    - The main class is in src/ThePackageThatContainsTheMainMethod/MainClass.java
      * Shouldn’t be too hard to find.
* Controls
  + Keyboard
    - Up arrow
      * Move your ship in the direction it is facing
    - Left arrow
      * Rotate your ship anticlockwise
    - Right arrow
      * Rotate your ship clockwise
    - Down arrow
      * Teleport your ship forwards in the direction it is facing
        + This will bring your ship to a standstill!
    - Space bar
      * Shoot
        + Will fire a bullet in the direction your ship is facing
      * Start game
        + Pressing the space bar when you are at the main menu will just start the game
    - B
      * Deploy a bomb
        + You can only deploy a bomb if you have a bomb in reserve.
        + You get a bomb whenever you finish a level, up to a maximum of 3 in reserve
        + Only one bomb may be deployed at a time
    - Escape
      * Pause, bring up the quit game prompt
  + Mouse
    - You can select options on the main menu by putting your mouse cursor over the text for the option and then clicking any mouse button.
    - This can also be used to select an option after bringing up the quit game prompt, or on the score recording prompts.
  + Keyboard and Mouse
    - Pressing any button
      * The Any Button
        + Respawn after you die
        + Start recording your score after you get a game over
        + Skip the opening title credits
        + Exit the full scoreboard view
* Super Helpful Informative Tips!
  + The titular Inconvenient Space Rocks
    - These come in three sizes
      * Big
        + 24 points for destroying these

Worth up to 36 points each

* + - * Medium
        + 4 points for destroying these

Worth up to 6 points each

* + - * Small
        + 1 point for destroying these
    - Decay
      * If you don’t destroy them, the big and medium asteroids will destroy themselves.
        + If they destroy themselves

They will spawn 5 asteroids of the next size down

* + - * + If you or an enemy destroy them

They will only spawn 2 asteroids of the next size down

* + - * Small asteroids do not decay.
        + They also don’t spawn any children when destroyed, which is nice
      * Working out how long it will be until the asteroid self-destructs
        + Big asteroids

Turn yellow as they get closer to self-destruction

* + - * + Medium asteroids

Turn red as they get closer to self-destruction

* + - * You can score more points by shooting an asteroid before it self-destructs (and then shooting the 2 new asteroids) than shooting the leftovers after it self-destructs
  + Points; why you want them, and how to get them
    - Reasons to try scoring points
      * To flex on everyone else when you beat everyone else’s score on the leaderboard
      * You’ll also get an extra life every time you score 100 points and that’s nice
    - How to score points
      * Destroying things
        + Destroying asteroids
        + Destroying enemy ships
      * How to destroy things
        + Shoot them
        + Blow them up with a bomb

You will only score half the normal amount of points for using a bomb to destroy something

* + - * + Bait an enemy ship into getting hit by an asteroid

You will not score points for this, but it will be very funny

* + - * + Ram into it

You will die.

But you will still score points.

And you will get an extra life if it takes you over the 100-point extra life threshold, even if you were about to get a game over.

* + Delaying the inevitable; how to postpone getting a game over
    - There is no end to this game.
      * You can’t win, but there’s plenty of opportunities to lose.
    - You get a game over when you run out of lives
      * You only start with 3 lives
    - How to actually delay the inevitable
      * Don’t die
        + Don’t crash into any asteroids
        + Don’t crash into enemy ships
        + Don’t crash into bullets fired by enemy ships
        + Don’t get caught in the explosion of your bomb
      * Score enough points to have a lot of extra lives in reserve
        + Remember; 100 points = 1 more life
  + Progression
    - After you destroy all the currently active asteroids, you have completed the level
      * You will get an additional bomb (up to a maximum of 3 bombs in reserve)
      * The asteroids for the next level will start to spawn in
      * The maximum amount of time it takes for enemies to spawn in will be decreased
    - You do not need to destroy all active enemies to move to the next level
    - Every third level, the limit on the number of active enemies will be raised by 1, up to a maximum of 5 active enemies at any given time

# Design choices, tuning parameters, testing, etc

* The overall presentation (why it uses TexturePaint and not sprites, and the ‘aware that it’s bad’ aesthetics)
  + In short, all of these design choices came from a desire to avoid any potential roadblocks in regards to plagiarism, by simply not using any assets obtained from somewhere on the internet.
    - The use of TexturePaint instead of sprites
      * In general, sprites should use transparency, so, when rendered, there isn’t an unsightly white box around the sprite you’re trying to use. However, when you don’t have access to an image editor that supports transparency (for example, when you’re on campus, and only have Microsoft Paint), you can’t really make sprites.
      * I found out about Java’s Graphics2D TexturePaint (this was a few weeks before it got mentioned in a lecture), which pretty much solved the issue I was having (as this can effectively be used to provide the illusion of a sprite being used), and, as I already used bounding boxes as part of the collision detection process, these combined with the textures as BufferedImages was all I needed in order to get the TexturePaint to render.
      * I did have to improvise a little bit for the ships. As their bounding box does tend to squash/stretch as they rotate, and having those deform very visibly was uncomfortable to view and could confuse a player, I instead used a defined Rectangle for them (updating with the position of the Ship) to counter the deformities. Their textures also have rotational symmetry (or as close as is possible with Paint), to hide the fact that these don’t rotate with the ship.
    - The remainder of the overall presentation
      * Basically, Microsoft Paint isn’t known for being a tool that can be used for amazing works of art. Therefore, I felt that I had carte blanche for ‘intentionally bad programmer art’ aesthetics.
        + The choice of using comic sans for all the text in the game, the very low-budget DIY sound design, and the rambling, non-pretentious nature of the opening cutscene (if it can be called a cutscene) was an extension of this.
        + It might also be related to the reasons for why I was prescribed antidepressants towards the end of term, I don’t really want to go into details, but in all honesty, with everything that’s been going on this month, along with the fact that this isn’t CE217, I’m probably going to refrain from any overly deep analysis of what it is I was thinking a few weeks ago.
* Having the game focus on entirely one screen
  + I tried a few experiments with viewport scrolling (keeping an xScroll and yScroll value in Game, obtained from the position of Ship in Game in terms of distance from the middle, and then translating everything by that amount when updating game (and then updating xScroll and yScroll so they would be reset), but they didn’t really work out, and so I eventually decided to cut my losses and just focus on the single screen.
* How the enemies were implemented
  + In short, the ship was implemented already, and it was just a case of getting an abstract class from the existing ship and bullets, and making Enemy and Player subclasses of those abstract classes
    - The player ship/bullet stuff were left as is
    - The enemy ship/bullets had a few changes
      * Different colours and texture
        + So you can tell them apart from the player ship/bullet
      * No respawn/reward invincibility for the enemy ships
        + As this invincibility stuff is mainly intended to prevent the player from dying in a ‘cheap’ way, so giving it to the enemies seemed a bit like a bad move
      * Enemy ships have a slightly slower turning speed to the player ships
        + The enemy ship AI (specifically the behaviours that take the player’s location into account) try to turn towards the player, however, they kept overshooting and having to re-adjust themselves, so they turn a little slower to make it harder for them to oversteer.
      * Enemy ships cannot deploy bombs
        + The bombs were a bit of an afterthought, and I figured that letting the enemies use them could be a little bit overpowered.
  + The enemy AI
    - All of the possible enemy behaviours are kept within EnemyController. This implements the Controller interface (similar to PlayerController), but all of the behaviours are stored as methods within EnemyController, and, whenever an enemy ship is revived, it calls the revive method if its EnemyController, which in turn makes it pick a behaviour to use until it is next revived, through a combination of Math.random() and a switch case statement to a method with the actual behaviours
    - The behaviours involve
      * Spinning around and shooting, thrusting after it goes slower than a certain speed
        + There is a boolean with a 50% chance of being true/false (set with Math.random() in the revive method) which is used to determine whether it spins clockwise or anticlockwise

This does not take the player’s position into account

* + - * Pursuing the player, shooting the player when it gets close
        + Obtains player position, thrusts until within a certain distance of the player, shoots the player when the player is within a certain distance and a certain range from the enemyShip’s angle
      * Pretending to be a turret
        + Not moving, but turning towards the player and shooting when the player is close
      * Not being a completely stationary turret
        + Same as above but thrusting after the speed falls below a certain threshold
      * Attempting to teleport into the player
        + Not moving or shooting, but constantly trying to turn towards the player’s ship and teleport into it
      * Moving and spinning whilst shooting
        + Like the first action, but starts spinning whenever it engages its thrusters, spinning in a different direction each time, and stopping spinning after a random period of time after it starts

This does not take the player’s position into account

* + - * Random inputs
        + All actions performed at random

Does not take the player’s position into account.

* Using ‘revive()’ methods for objects instead of constructing them whenever they are used
  + At first, every single object was constructed whenever it was to be used in the game, and every GameObject would have an ArrayList of any child GameObjects constructed by it, and these would be added to the Alive list (and this child object list would be cleared) if it had anything in it in an update loop
    - However, this wasn’t very efficient, as everything had to be declared when needed and garbage-collected when no longer needed, with a single big asteroid requiring the creation of, in total, between 6-30 additional child objects after getting destroyed, and these objects would also need to be garbage collected at some point
    - And there’s also the infinite number of bullets that a ship could spawn in
  + Moving them into a stack, and reviving them when needed, avoided this problem, as they were only created at the very start of the model being executed, and no more GameObjects would need to be created/garbage collected during runtime
    - However, certain object attributes of them (such as any changing colours, polygons, bounding boxes, Vector2Ds, etc) are still created, destroyed, and need garbage collection throughout runtime, so there is scope to improve this
    - The limit on the number of allowed objects allows each GameObject to have roughly 3 ‘children’ active
      * 5 EnemyShips can have a total of 15 EnemyBullets
      * 15 BigAsteroids can break into 50 MediumAsteroids
        + 3.3 recurring potential children
      * 50 MediumAsteroids can break into 150 Asteroids
      * 1 Bomb and 4 PlayerBullets exist
      * Only one PlayerShip exists, but this is stored as a variable, not in a stack
    - In the update loop, if an object is dead, instead of being updated and put in the ‘alive’ list, it will be pushed back into the appropriate stack
    - If a GameObject is trying to spawn a certain child object, but the stack for its children is empty, it will not spawn that child
      * However, they will only attempt to spawn in child objects after all of the dead objects have been pushed back into the stack within the loop through all of gameObjects
        + The only exception to this are the EnemyShip objects, but no sane player will complain about an enemy not being able to shoot at them when it should have shot at them. This is because having a separate iteration through the Alive list for the sole purpose of finding EnemyShips which can shoot and then making them shoot would have added too much overhead to be justifiable.
        + The PlayerShip can be accessed as a variable, and the dead asteroids are found when it iterates through the dead list to work out what to do with each of the dead items before discarding them
  + New objects are also spawned in via the ‘revive’ method upon starting a new level
    - A new object is added to the level every other update call at the start of a level, not for any performance related reasons, but just because I felt that having the asteroids appear one at a time looked better
      * Adding them every other update call also makes it possible for the player to see them appearing one at a time (game runs at 50fps, human eye can only do ~24fps, so spawning them in at 25 asteroids/second can actually be perceived)
  + This was tested by trying to spawn in as many objects as I could in the game (via a custom level config that tries to spawn in well over the maximum number of allowed asteroids), and seeing how it reacted
    - This code is still present in the game as an easter egg. If you hold ‘p’ as a level starts, this secret level will start instead. And the game still runs rather smoothly.
* The way that the levels were stored
  + The game I made for my ce203 assignment involved levels with varying defined quantities of 3 (well, 4, but the 4th was effectively the default) different types of objects, so I pretty much used the exact same approach for this game, except storing quantities of each size of asteroid.
    - On that note, the HighScoreHandler and ScoreRecord classes were also re-purposed from that (but the JOptionPanes now have html formatting, it can export the leaderboard as an arrayList of Strings, and has ScoreRecord as an inner class)
  + After you finish the 8th level, the game will re-use the level 8 configurations, but adding a random number to each asteroid’s quantities, up to a maximum of 10 potential additional asteroids of each type (but the defined asteroid stack sizes still cap the number that can actually be stored)
* Balancing out the points
  + I wanted to incentivize players to shoot asteroids before they destroyed themselves, and I think that the way points are awarded fairly rewards players for doing this.
    - Small asteroids
      * 1 point each
    - Big/medium asteroids
      * Award you with the same number of points as you would have got from destroying 4 of their child asteroids (and children), allowing you to potentially score up to the value of 6 child objects
        + Big: 24 (+12, up to 36 points)

12 potential points from the spawned medium asteroids

* + - * + Medium: 4 (+2, up to 6 points)

2 potential points from the spawned small asteroids

* + - * If you don’t destroy them in time, your maximum potential score is lower, and there’s also more asteroids to deal with
        + Big:

Up to 30 points from the medium asteroids

Up to 25 points if medium asteroids are allowed to expire

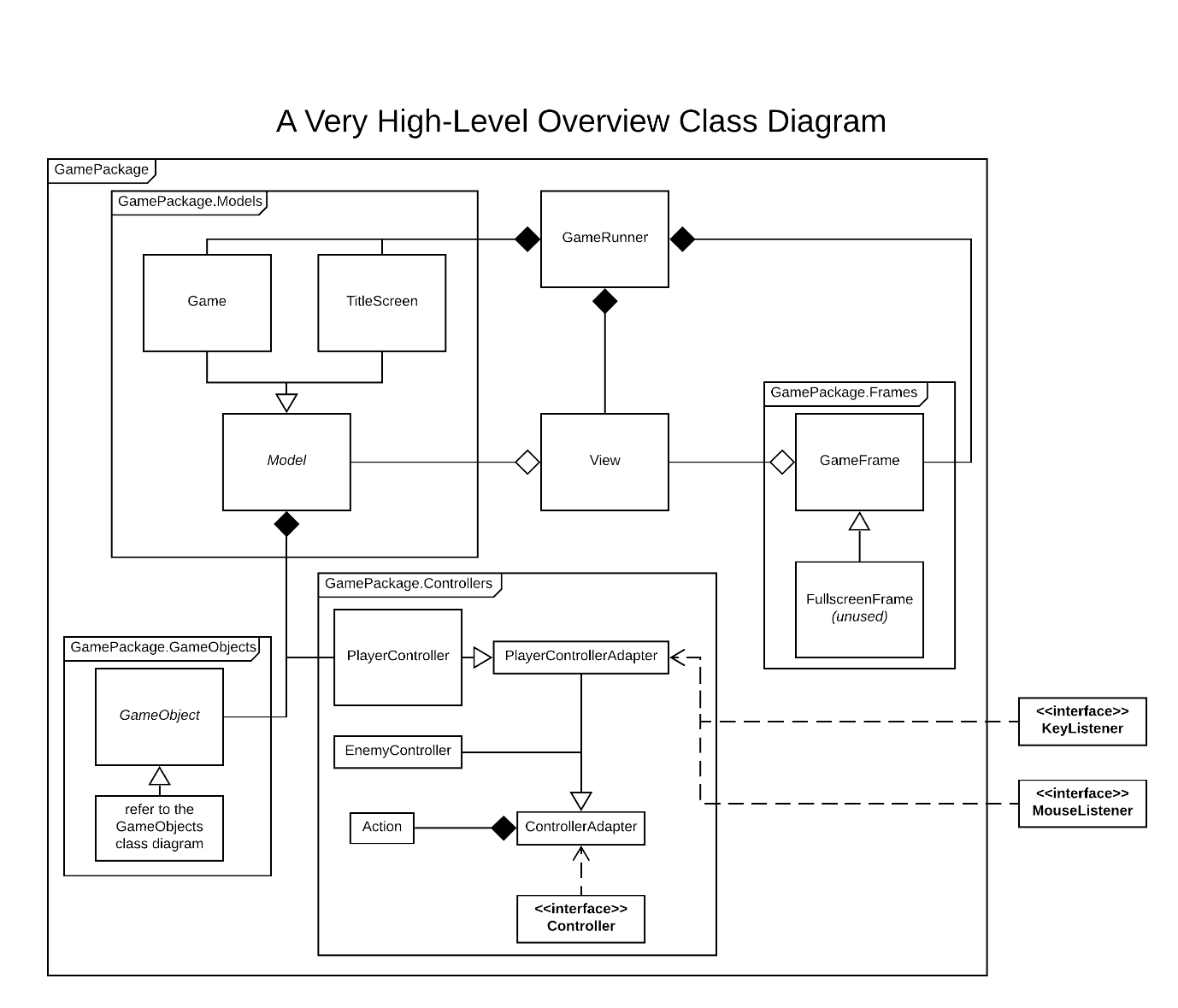
* + - * + Medium

Up to 5 points from the small asteroids

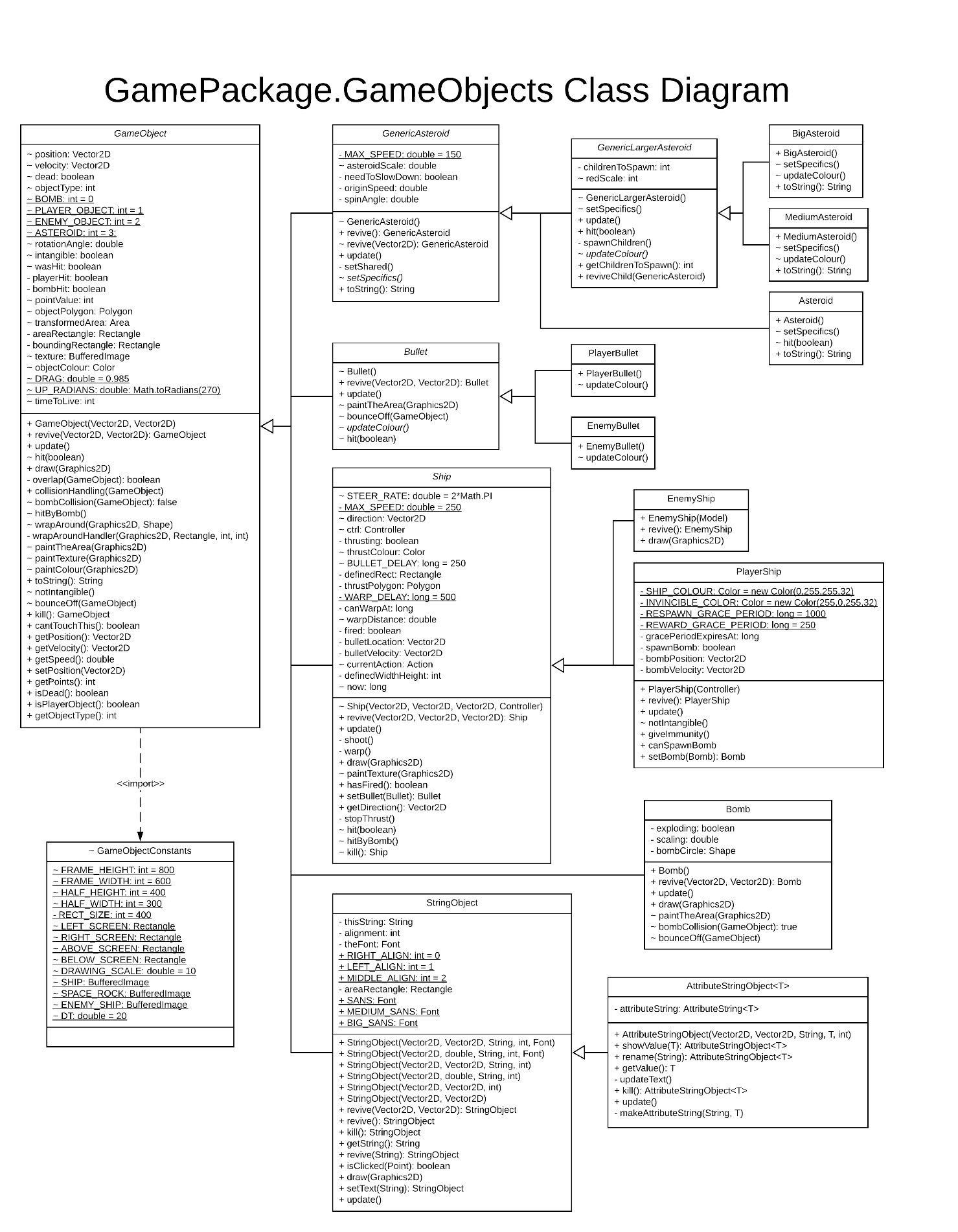
* Giving the player some idea about time to live
  + In short, Java can use colours with RGB values between 0-255, and it was just a matter of using the timeToLive attribute of certain gameObjects to modify the red/green/blue values of these colours appropriately, so the player can, at a glance, work out how long an asteroid has before exploding/a bullet has before forgetting how to exist
* Coloured overlays for the textures
  + The textures are rendered via TexturePaint and graphics2D.fillArea(transformedArea), so it was just a matter of giving the GameObjects a Color object with some transparency, and, after drawing the texture (if needed), changing the Graphics2D paint to that colour, and then filling in the same area to give the texture a coloured overlay.
* The HUD/other text stuff
  + Graphics2D’s drawString() method allows a string to be drawn at an arbitrary place/position on a screen, so I could have just used that in View, constantly calling Game for the score/lives/level/bombCount
  + However, encapsulating these as a StringObject GameObject allowed some more versatility with these, such as easily making the font comic sans, allowing them to be shown/hidden via killing/reviving them, changing the string they show, changing their alignment, changing their positions, allowing them to be clicked, letting them move (which is how I was able to pull off the scrolling text, killing them as soon as they went offscreen (negative y position)), etc.
    - The AttributeStringObject<T> was the next logical step from the AttributeLabel class I created for my CE203 assignment 2 program, obtained by generifying it from a JLabel with a String and an int to instead be an AttributeString<T> effectively combining a String and a generic type, and then wrapping this generified class in a subclass of StringObject.
      * This class exists because it made it easier to update the bomb/score/lives/level counts shown to the user.
  + All of the StringObject/AttributeStringObject<T> objects are always kept in separate hudObjects/aliveHUD lists, away from the GameObjects
    - This is so the View can render hudObjects above all of the GameObjects in gameObjects, by rendering all of gameObjects before rendering the hudObjects, ensuring they are always visible at all times.
  + The intro text/full leaderboard scrolling text are effectively identical in how they are implemented (except the leaderboard has some title text at the start and scrolls faster), as they’re a lot of StringObjects that are created from an ArrayList of Strings (both of which coincidentally are constructed using the contents of a .txt file), with constant upwards velocity, and being removed when off-screen, going back to the main menu either after they finish or the player presses anything.
* Swapping models in View, to switch between a title screen and the game itself
  + How it works
    - When the game is launched, and the view is eventually displayed, there is no model within the View, only a background to say that the game is loading (and no model), whilst the GameRunner constructor is initialising everything else (including the TitleScreen and Game models)
    - When everything is set up, and the GameRunner constructor finally reaches the MainLoop function, it is essentially told that the Game is currently active, and must immediately revive ‘TitleScreen’, pass that to View, and that it should now update it.
      * After this is done, it starts playing the appropriate music, starts the RepaintTimer for making the view repaint itself (and the components in the Model), and commences updating the TitleScreen model
      * After the TitleScreen model’s endGame attribute is set to true (signifying that it is done), it is swapped out for a revived Game in a similar way, swapping back to a revived TitleScreen again after the game is over, and so on until the player quits
    - Similar principle to swapping out game cartridges/game discs in a games console. Same thing being used to run it, things being run are identical at a high level, but their low-level contents are completely different.
  + View is also generified so it contains a Model (not a Game or TitleScreen), and the stuff for drawing the gameObjects and hudObjects is synchronized with Model.class instead.
* Having the EscapeListener as its own class, unrelated to PlayerController, despite both being concerned with keyboard inputs
  + All of the Controllers are concerned with model-level inputs, for controlling the GameObjects within the models.
  + However, the EscapeListener concerns the quit prompt, which is on a higher level than the Model level, as this needs to interrupt the Model’s updates (and the View’s repaints) in order to properly pause the game efficiently, as, if this was integrated into the controller, it would have meant that the model would still have continued ‘update’ calls, even if the contents may have been skipped via a ‘break;’ if statement, whilst simply pausing all the updates and repaints entirely is generally easier to deal with.
* The asteroids and enemy ships in the background of the TitleScreen
  + These can run perfectly well without the player being present (the EnemyControllers are given the position of a random GameObject/a random vector when they request the player’s position from the Model), therefore, I just edited their spawning behaviour so EnemyShip/BigAsteroid objects could be revived from the stack (if possible) at random intervals, spawning in EnemyBullets/other asteroids from their stacks when needed, and it both works for giving players something to look at on the title screen, and can be used to demonstrate the enemies doing their thing.
* The choice of bounding boxes and then area intersections for collisions
  + I used this approach because the bounding boxes were useful for the texture paint stuff, and then, as I was using defined areas which were easily visible to the players, and not sprites, I figured that two stage collision detection (checking bounding boxes and then the actual areas) was appropriate to get pixel-perfect collision stuff
    - I attempted to limit the collision detection so it would only check objects within a certain radius, however, as it wasn’t very easy to test the method I had made for distances between two vectors with a wraparound (as g.drawLine can’t really be wrapped around), and due to the computational time involved with this calculation, I stuck to the two-level collision check thing I had.
    - I also didn’t have much confidence in the collision cell approach for this game, as the wraparounds probably would have caused a fair share of issues with it, and, even on my very low-end laptop, even when the maximum number of asteroids are active, the game still runs smoothly with the approach I used.

# Class Diagrams

## Overview



## GameObjects Class Diagram



# Appraisal of achievements

* What went well
  + The game works
    - It compiles
    - It can be run
    - It has sound
    - It has graphics, both in the form of coloured shapes and in textured shapes
    - No assets had to be sourced from a third party
    - It can record your high scores
      * Persistent record in a .txt file
    - It can show the high scores to anyone who plays it
    - The additional constraint for this year’s assignment (asteroids must break into more child asteroids than usual if not shot within a certain period of time) has been fulfilled
    - Game objects can wrap around the playable area
      * The hitboxes/textures of everything still match up to the object itself when it wraps around the playable area
    - You can actually play the game
    - There are asteroids and enemy ships (which fire enemy bullets) which add some challenge to the game
    - You can shoot to defend yourself/attack/get points
    - The levels are randomly generated
      * Random premade configurations for first 8 levels, and then randomized variation on a premade configuration after that
    - There are a couple of easter eggs thrown in
      * ‘secret level’ by pressing p
      * Some other secrets happen when clicking on the title/subtitle text in the main menu
    - You can earn and use bombs as you advance through the game
      * Risk-reward element to these as well – lethal to everything, but anything can destroy it when hit before it explodes
    - All collisions are pixel-perfect
      * Hitboxes match up exactly with how they were rendered to the user by the view; no complaints about disjointed hitboxes or anything like that
        + Even works when the objects wrap around the screen
* What didn’t go well
  + Implementation of advanced game features
    - The game is still constrained to a single 800px by 600px area
      * Was not able to get viewport scrolling stuff implemented
    - Was not able to implement fullscreen support
      * If you attempt resizing the window, it won’t reset itself to the appropriate size, which is somewhat unsightly
      * Some code is here to facilitate this being added, but it would require a lot of refactoring
    - No ‘power-ups’ were implemented
      * Whilst you do get temporary invincibility after destroying something, and a bomb after finishing a level, there aren’t any traditional power ups (no objects that you can simply ‘pick up’ and get some benefit from)
    - Nothing involving the use of sprites is present (no sprites, no parallax background stuff, etc)
      * Sure, I didn’t really have the tools to create a sprite, but this still is a bit of a detriment to the overall visual appeal
      * I probably could have used TexturePaint to simulate a sprite for some parallax background stuff, but I didn’t get around to doing it
    - No visual distinction between EnemyShips with different behaviours
      * They act differently, but they all look exactly the same, and are all still functionally identical
    - The hitboxes are only updated in the draw thread
      * This does mean that the collisions only happen if you have seen them collide (as, if a tree falls in a forest and there’s no-one around to see it, does it perform a very epic backflip?), however, it also means that all GameObjects must remain intangible after being revived, until they first get drawn, to avoid having to throw/catch a NullPointerException when trying to run collision detection on them.
      * The pass-through problem is somewhat present
        + There isn’t really any way to stretch an area in Java (apart from getting the union of a prior area and an after area), so I’m not sure how I could easily fix this
  + This document
    - I honestly wasn’t sure what was expected for this document, but I mainly wanted to get this done, haven’t had any chance to process everything else that’s been happening recently because I’ve been trying to get this done first.
* Things that could be developed further
  + The efficiency of ‘reviving’ objects
    - Most Object attributes of many objects which are revived (most notably the stacks in the Model objects) get completely wiped and re-initialized as part of the revive method of the object holding them, therefore, it would be probably be worth defining a constant within these revived objects to hold a default value (if appropriate) for any non-primitive attributes, so, when the object is revived, this would cut down on the amount of initialisation which would have to be performed
  + The strings being displayed to the user
    - Instead of having these hardcoded in, I could have instead stored them in separate localisation text files, where, similar to how the HighScores file stores player names and scores, it would instead contain identifiers and strings, so, when the user starts the game, they would be prompted to choose a language, then a HashMap would be constructed using that file (associating identifier strings to the strings for that language), and, when a string is to be displayed to the user, it would just obtain the string for the associated identifier from this HashMap.
      * I don’t really have time to implement it at this point, unfortunately.
  + Not using any Constants files
    - I attempted to get the constants more modular (so everything in each package would only have access to the constants that it needed), but I should have tried harder to fully refactor these out, and, if I was able to successfully do that, it may have made it easier to implement the fullscreen stuff, as there wouldn’t be any omnipresent FRAME\_HEIGHT or FRAME\_WIDTH constants in the way or anything like that.
  + The package structure
    - There’s still an unfortunate amount of methods/attributes within the GamePackage package with public modifiers, and I guess that if I had improved the package structure I had within it (or if I had not tried to create separate packages for the Models, Controllers, and GameObjects in the first place), I could have potentially solved this problem. But, again, time constraints.